# Git and GitHub

**CHARACTERISTICS:**

* ***Distributed Version Control Systems (DVCS):***

A distributed [version control system](https://about.gitlab.com/topics/version-control/) (DVCS) brings a local copy of the complete repository to every team member’s computer, so they can commit, branch, and merge locally.

1. Fork and copy options make it DVCS.
2. Otherwise, all basic characteristics like it also keeps track of changes made to a particular project or file.
3. means that users anywhere in the world can have a copy of your project on their own computer.
4. When they’ve made changes, they can sync their version to a remote server to share it with you.
5. Version control systems are widely used for things involving code, but you can also version control images, documents, and any number of file types.
6. Collaboration is most important feature of git. As via this we can collaborate with one of our team members who lives other part of world.
7. Linus Torvalds made it in 2005.

**[ Fork repository** --> **make changes** --> **push changes** --> **pull request** --> **merge changes ]**

***TERMINOLOGY :***

* *SSH PROTOCOL* : The SSH protocol is a method for secure remote login from one computer to another ( like server is also computer).
* REPOSITORY : A repository contains your project folders that are set up for version control.

Basically it’s a data structure for storing documents including application source code.

  A repository can track and maintain version-control.

* FORK : A fork is a *copy of a repository*.
* PULL REQUEST : A pull request is the way you request that someone reviews and approves your changes before they become final.
* WORKING DIRECTORY : Directory in which we are working.
* GitHub : It is the online hosting service for Git repositories.

***BASIC GIT COMMANDS:***

* *“ git init “ intializes a directory as a git repository.*
* "git add" moves changes from the working directory to the staging area.
* "git status" allows you to see the state of your working directory and the staged snapshot of your changes.
* "git commit" takes your staged snapshot of changes and commits them to the project.

* "git reset" undoes changes that you’ve made to the files in your working directory.
* "git log" enables you to see previous changes to a project.
* "git branch" lets you create an isolated environment within your repository to make changes or to work on different update/ feature.
* "git checkout" lets you see and change existing branches.
* "git merge" lets you put everything back together again. Or merges the branch.

**Basic vim commands:**

* Type vim or vi in git bash this will open another tab of vim editor.
* In order to get typing press i (stands for insert) this will bring insert mode.
* When you are done typing press ESC to exit from insert mode.
* Then press :x for saving your work and exit from vim editor.

### BASIC GIT COMMANDS FOR CREATING FILES AND FOLDER:

1. ‘touch’ command is used to create new file.
2. ‘mkdir’ command used to create new folder.
3. ‘ls’ command is used to list all the files and folders in that directory.
4. ‘ls - a’ command used to list all the files or folders even if they’re hidden.
5. ‘pwd’ command = present working directory, so it tells present working directory.
6. ‘cat’ : this command lets you see content of a file.
7. ‘echo’ this command appends something to a file

Ex. *echo debug.log >> .gitignore*

*Here : ‘debug.log’ is something we’re appending*

*And ‘.gitignore’ is file.*

# Ignoring files

1. **Command to untrack a file :**

git rm --cached <filename>

1. **Command to untrack all files in one command :**

git rm -r --cached . (don’t forget this point).

**Setting up a git repository**

There are two ways to setup a git repo:

1. Using git init
2. Using git clone

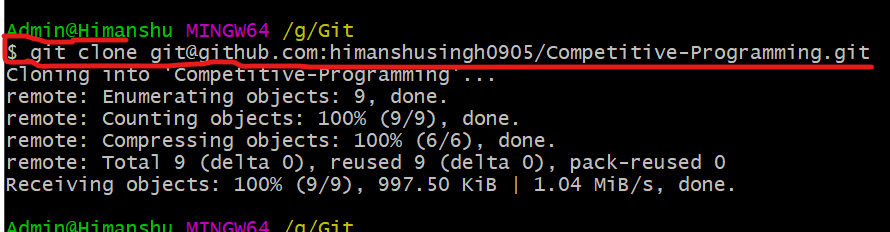
git init :

* To create a new repo, you'll use the git init command.
* git init is a one-time command you use during the initial setup of a new repo.
* Executing this command will create a new .git subdirectory in your current working directory.
* This will also create a new main branch.

git clone:

* git clone setups remote repository in your local computer.

Syntax : git clone < url >



**git config**

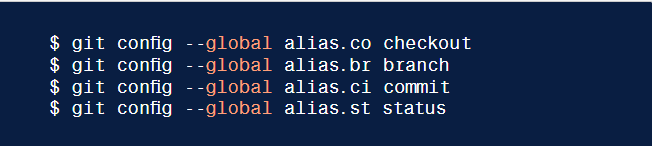
* You can view all of your settings and where they are coming from using:

*$ git config --list --show-origin*

* *$ git config --global user.name "John Doe"*
* *$ git config --global user.email johndoe@example.com*
* *$ git config --global core.editor emacs*
* *$ git config --global init.defaultBranch main*
* *If you want to check your configuration settings, you can use the git config --list*

# Git Alias

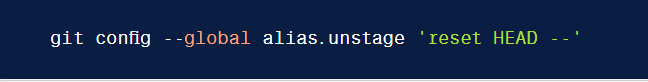
**Examples :**

****

Here , co = alias of checkout

br = alias of branch , similarly for others

**Example(2) :**



# Saving changes

**[git add](https://www.atlassian.com/git/tutorials/saving-changes),**  **[git commit](https://www.atlassian.com/git/tutorials/saving-changes),**  **[git diff](https://www.atlassian.com/git/tutorials/saving-changes),**  **[git stash](https://www.atlassian.com/git/tutorials/saving-changes)**  **[.gitignore](https://www.atlassian.com/git/tutorials/saving-changes)**

1. *git add  :*

*command adds a change in the working directory to the staging area.*

*Alternatively we can use other git add commands:*

*git add . <- it adds all files at a time.*

*git add -a <- it adds all files at a time.*

*git add <file\_name> <- only adds specific file.*

1. *git status to view the state of the working directory and the staging area.*

# Git commit

1. *git commit :*

captures a snapshot of the project's currently staged changes.

*git commit -m "commit message"*

Using this command we can skip a step of writing message.

*git commit -a-m "commit message"*

*Using this command we can directly commit modified files but it shouldn’t*

*Be a untracking file.*

*git commit --amend*

Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit.

## Comparing changes with git diff

By default *git diff* will show you any uncommitted changes since the last commit.

1. git diff
2. git diff --color-words

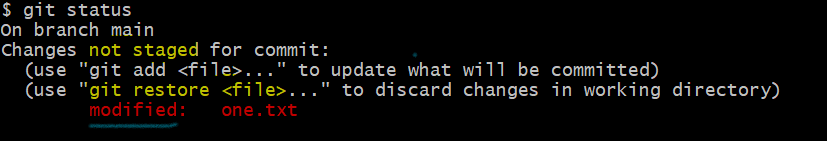
# Undoing Commits & Changes

Basically there are four different sections a file can go through:

1. Un-tracked : (files which are never staged)
2. Staged : (files which are staged with green color)
3. Un-staged/ Modified : ( Files which once staged but later may be modified so comes in this section)
4. Commited : (files which are commited)

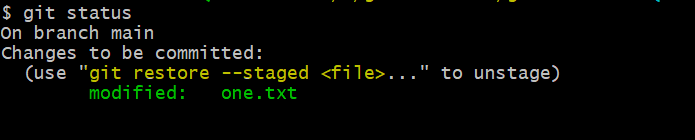
# To remove changes from a modified file i.e. file which is in Un-staged area.

Command : git restore <file Name> this command will unstage this file from working tree, It means it will discard all changes made to that file after last commit.



# To remove changes from a staged file or To unstage a file

Command : git restore --staged <file Name>



See this file is currently in Staged area so unstage it above command is used.

# Git Branch

1. git branch <branch Name> For creating a branch
2. git checkout <branch Name> For switching to another branch from current branch
3. git checkout -b <branch Name> shorthand of above two commands
4. git branch -d <branch Name> for deleting branch
5. git merge <branch Name> this command will merge branches

PUSHING A WHOLE BRANCH TO REMOTE :

Command : git push <remote> <branch>

Ex. git push origin develop

**# Configuring ignored files for all repositories on your computer :**

* You can also create a global .gitignore file to define a list of rules for ignoring files in every Git repository on your computer.
* The rules written in this file are applied to all git repositories.

1. Open Git Bash.
2. Configure Git to use the exclude file ~/.gitignore\_global for all Git repositories.

$ git config --global core.excludesfile ~/.gitignore\_global

**RULES :**

* ***Git actually ignores patterns. (very deep statement)***
* ***And git actually track files not folders.***

|  |  |  |
| --- | --- | --- |
| debug.log | debug.log logs/debug.log | By default, patterns match files in any directory |
| debug?.log | debug0.log debugg.log but not debug10.log | A question mark matches exactly one character. |
| debug[0-9].log | debug0.log debug1.log but not debug10.log | Square brackets can also be used to match a single character from a specified range. |
| debug[01].log | debug0.log debug1.log but not debug2.log debug01.log | Square brackets match a single character form the specified set. |
| debug[!01].log | debug2.log but not debug0.log debug1.log debug01.log | An exclamation mark can be used to match any character except one from the specified set. |
| debug[a-z].log | debuga.log debugb.log but not debug1.log | Ranges can be numeric or alphabetic. |
| logs | logs logs/debug.log logs/latest/foo.bar build/logs build/logs/debug.log | If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named logs are ignored |
| logs/ | logs/debug.log logs/latest/foo.bar build/logs/foo.bar build/logs/latest/debug.log | Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and  subdirectories – will be ignored |
| logs/ !logs/important.log | logs/debug.log logs/important.log | Wait a minute! Shouldn't logs/important.log be negated in the example on the left  Nope! Due to a performance-related quirk in Git, you can not negate a file that is ignored due to a pattern matching a directory |
| logs/\*\*/debug.log | logs/debug.log logs/monday/debug.log logs/monday/pm/debug.log | A double asterisk matches zero or more directories. |
| logs/\*day/debug.log | logs/monday/debug.log logs/tuesday/debug.log but not logs/latest/debug.log | Wildcards can be used in directory names as well. |
| logs/debug.log | logs/debug.log but not debug.log build/logs/debug.log | Patterns specifying a file in a particular directory are relative to the repository root. (You can prepend a slash if you like, but it doesn't do anything special.) |